AFRL-IF-RS-TR-2005-273
Final Technical Report
July 2005

CLIENT SECURITY IN SCALABLE AND SURVIVABLE OBJECT SYSTEMS

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. N301

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-273 has been reviewed and is approved for publication

APPROVED: /s/
ALAN J. AKINS
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 2005	**Final** **Jun 02 – Jun 04**

4. TITLE AND SUBTITLE

CLIENT SECURITY IN SCALABLE AND SURVIVABLE OBJECT SYSTEMS

5. FUNDING NUMBERS

G - F30602-02-2-0114
PE - 62301E
PR - N301
TA - FT
WU - N1

6. AUTHOR(S)

Michael Reiter

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Carnegie Mellon University
5000 Forbes Ave
Pittsburgh PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2005-273

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Alan J. Akins/IFGA/(315) 330-1869 Alan.Akins@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

This report describes an architecture for defending against client capture in a survivable distributed object store called Fleet. The work was primarily concerned with the case in which clients accessing objects are user-driven devices that should be rendered unusable if taken from their rightful owners, but yet are otherwise authorized to invoke methods on objects. Toward this end, we integrate a technique called “capture resilience” into the Fleet system. We demonstrate that capture resilience has a symbiotic relationship with Fleet: in addition to hardening Fleet against client compromise due to physical capture, the capabilities that Fleet offers permits the construction of a capture protection infrastructure with better properties than were previously attainable. This infrastructure is the primary focus of this document.

14. SUBJECT TERMS

Fault tolerant networks, DDoS, denial of service attack, cyber defense, capture resilience, Fleet, client security

15. NUMBER OF PAGES 19

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1 Introduction

The pervasiveness of security vulnerabilities in commercial off-the-shelf (COTS) computer systems has prompted much research in building *survivable* (or *intrusion tolerant*) distributed services. In this approach, COTS systems are composed to implement a distributed service that is robust to successful attacks on these individual components. The research literature has documented significant strides in the development of such services (e.g., [21][22][4][12]), and we ourselves have constructed software to implement such distributed services in a previous DARPA program, namely the Fleet survivable object store [16]. Despite the successes described in the research literature, significant obstacles remain to the deployment of this approach on a wide scale.

The high-level goal of this research program was to address what we perceive as the most challenging obstacle, namely vulnerability to client compromise. As a simple example, consider a distributed service that implements the abstraction of shared files, and does so "survivably" in that it masks the corruption of individual file servers from clients. Despite this, if a client with authority to write to a file is compromised, then this client can arbitrarily overwrite the file, effectively corrupting every server and rendering the file useless to the application that requires it.

Our goal in this research was therefore to extend the Fleet system to include defense against corrupt clients. Our efforts focused on clients that are driven by a human user, and that should be disabled if the client device falls out of physical possession of that user. This is a category of client that is only becoming more important, especially with the widespread deployment of mobile devices such as programmable mobile phones and PDAs, and with the anticipated deployment of wearable computing devices. Indeed, if our experience with laptop computers and mobile phones is any indication, then these devices will be stolen frequently. And, the importance of defending against captured wearable computers in battlefield situations should be obvious.

To defend against captured clients, we applied techniques we have developed that permit the client device to perform cryptographic operations (e.g., to digitally sign a request to modify an object) only after the device has convinced a remote server, here called a *capture-protection server*, that the device is still in the possession of the correct user [14][15]. Our techniques are particularly powerful in that the capture-protection server is untrusted; even if compromised, it does not pose a threat to the cryptographic keys of the device (unless the device is also captured). These techniques are also powerful in that they permit a device to *delegate* from one capture-protection server to another, so that subsequently the second server is *authorized* to perform the capture-protection function for that device [15]. Using delegation, the device can ensure that it has a capture-protection server in relatively close proximity at all times, so as to minimize the latency of interacting with the server in the course of performing cryptographic operations.

The integration of these techniques with Fleet promises to significantly harden Fleet against this important class of threats. At the same time, however, it offers opportunities for improving our capture-resilience techniques. This potential is best understood by considering the specific function of a capture protection server, i.e., to confirm that the device remains in the possession of a legitimate user before permitting it to perform a cryptographic operation. Presuming a password is used to perform this confirmation (as in [14][15]), the server must limit the number of incorrect guesses against the device's password, lest it permit an attacker who has captured the device from progressing too far in an online dictionary attack. When servers are dynamically authorized, however, this may widen the window of vulnerability to such an attack: If the attacker captures the device, then it can mount an online dictionary attack against *each* currently authorized server, thereby gaining more password guesses than any one server allows. A second security challenge arises from the feature that a capture protection server can be *disabled* for a device if the device is captured, even if the attacker has compromised the user's password. Delegation also poses challenges to disabling: If the device and password are compromised, and if there is some authorized server when this happens, then the attacker can delegate from this authorized server to *any* server permitted by the policy set forth when the device was initialized. Thus, to be sure that the device can never use its key again, every server in this "admissible" set must be disabled for the device.

A proper solution to these problems would be for the capture-protection servers to coordinate among themselves, e.g., to inform each other of the incorrect password guesses that have been made by a device. A focus in this document is the design of such an architecture that supports secure data sharing among capture protection servers in a way that reverses the negative effects of delegation. As a result, the number of password guesses permitted against a captured device is unaffected by the number of servers authorized for the device, and disabling the device at one authorized server has the effect of disabling the device at all servers. However, with these benefits come significant costs to availability, in that the failure of *any* authorized capture-protection server can indefinitely prevent the device's use of its cryptographic keys.

Fortunately, using the capabilities of Fleet to build highly survivable capture-protection servers, we can largely eliminate this availability concern in the capture-protection infrastructure. Due to its implementation using Fleet, each capture-protection server will remain available despite benign and even malicious faults (hostile penetrations) of a fraction of its replicas, by virtue of the Fleet replication and replica coordination protocols. Fleet thus enables the adoption of this coordination architecture in critical applications, while simultaneously benefiting from it.

In addition to improving security, our capture-protection architecture exploits "locality of reference" by a mobile user, in two respects. First, our approach imposes communication overhead only when the device switches from using one capture protection server to using another; after one interaction to perform a cryptographic operation with the new server, there is no additional communication overhead for subsequent operations. Second, if delegation patterns follow a user's travels, the communication overhead of

switching servers is typically incurred only between the new server and the previous one; there is no need to retrieve data from a distant "home location" or a designated server.

2 Related work

Online services that play a role similar (though not identical) to that of a capture protection server include the modified Kerberos server of Yaksha [11], a *semi-trusted mediator* [2], and a *security mediator* in server-aided signatures [10]. These servers are interposed in the critical path of a user performing cryptographic operations using her private key, and thereby can disable a private key that should no longer be used. To our knowledge, no prior effort other than that from which we build [15] has proposed a notion of delegation from one server to another, and consequently the issues that we attempt to address here have not been previously considered in these other efforts.

At the basis of our capture protection infrastructure for coordinating capture-protection servers is a novel protocol for achieving mutually exclusive access to a mobile object. Our protocol was inspired by prior algorithms for similar goals (e.g., [19][18][4][8][1][20][25][3][6]), but at the same time differs from them in significant ways. First, we assume a dynamic network topology determined by delegation patterns, whereas most prior work on distributed mutual exclusion for mobile objects (e.g., [19][18][8]) builds on static topologies. Second, we permit Byzantine node failures [13] within our attacker models (a requirement for survivable systems), while most prior efforts in fault tolerant mutual exclusion for mobile objects deal with only benign node failures (e.g., [4][1][20][25][3][6]).

3 Background in capture protection

In this section we present background in capture protection, and then develop our coordination protocols for capture protection servers in Sections 4–6. To simplify the discussion in these sections, we will avoid discussion of Fleet-specific matters and presume that capture protection servers are non-replicated. We will return to the implementation of these techniques in Fleet (in which capture protection servers are implemented as replicated Fleet objects) in Section 7.

A capture protection system consists of a device dvc and an arbitrary number of computers called *nodes*, each denoted nd. Each node can host (execute) multiple logical capture protection *servers*. A server is denoted by svr, typically with additional subscripts or other annotations. In our system, the device is used for generating digital signatures[1] (e.g., using RSA [23]), and does so by interacting with one of the servers over a public network. The signature operation is protected by a password π. The system is initialized with public data, secret data for the device, secret data for the user (i.e., π), and

[1] The device can also be used for decrypting messages, however, for simplicity we only deal with signatures here, as this is the most pertinent in the context of Fleet.

secret data for each node. The public and secret data associated with a node are simply a certified public/private key pair for the node, which are assumed to be established well before the device is initialized. We denote the public key of a node nd by pk_{nd}, and its private key by sk_{nd}. Each svr has a public key pk_{svr} that is simply the public key pk_{nd} of the node nd executing svr.

The device-server protocol allows a device operated by a user who knows π and enters it correctly to sign a message with respect to the public key of the device, after communicating with one of the servers. The device is initialized with one server available to it, denoted svr* and executing on node nd*, though the device can cause a new server to be created on another node via delegation. For dvc to deploy a new svr on a node, another existing server svr′ must consent to delegating its authority to that node, after verifying that the creation of a server on that node is consistent with policy previously set forth by dvc (see below for details) and is being performed by dvc with the user's password. In this way, delegation is a protected operation just as signing is. The device can unilaterally *revoke* a server when it no longer intends to use that server. A node can be *disabled* (for a device) by being instructed to no longer respond to that device or, more precisely, to requests involving the device's key.

Here we will not specify a policy that determines the nodes to which dvc can delegate, here called the *admissible* nodes, though we do assume that the public key of such a node can be determined reliably. The policy that defines admissibility is user-tunable and must be set when dvc is initialized. An example policy might allow delegation to any node with a public key certified by a given certification authority. (Note this would also justify our assumption that the public key of an admissible node could be determined.) For such a policy, the admissible nodes are not known in advance and can change over time. Our approach is specifically designed to accommodate such flexibility.

Each attacker we consider controls the network; i.e., the attacker controls the inputs to the device and every node, and observes the outputs. Moreover, an attacker can permanently *compromise* certain resources. The resources that may be compromised by the attacker are any of the nodes, dvc, and π. Compromising reveals the entire contents of the resource to the attacker and permits the attacker to impersonate it. The one restriction on the attacker is that if he compromises dvc, then he does so after dvc initialization and while dvc is in an inactive state—i.e., dvc is not presently executing a protocol on user input—and the user does not subsequently provide input to the device. This decouples the capture of dvc and π, and is consistent with our motivation that dvc is captured while not in use by the user and, once captured, is unavailable to the user.

We formalize different aspects of the system described thus far as a collection of *operations*.

- dvc.delegate(svr, nd): dvc performs a delegation with server svr, using the correct password π, to deploy a new server on nd.
- dvc.revoke(svr): dvc revokes svr, indicating it will not be using svr in the future.

- nd.disable: nd stops responding to any requests from dvc (signing or delegation).
- dvc.comp: dvc is compromised.
- nd.comp: nd is compromised.
- π.comp: the password π is compromised.

We note that nd.comp compromises all servers ever hosted by nd. When convenient, we will use svr.comp to denote nd.comp where nd is the node hosting svr.

4 Overview of algorithms

Here we provide only the essentials of how the delegation and signature protocols of [15] work. The capture protection system requires a device initialization phase, for which the device dvc takes as input its private key sk_{dvc}, the password π, and the identity of node nd* with public key $pk_{\mathsf{nd}*}$. The output of initialization is a *ticket* τ* and an associated *authorization record* $\mathsf{authrec}_{\tau*}$ containing secret information for the device. τ* is a ciphertext encrypted under $pk_{\mathsf{nd}*}$ by dvc, the plaintext for which is generated as a function of π, a secret stored in $\mathsf{authrec}_{\tau*}$, sk_{dvc}, and an "inner ticket" ζ* that is itself a ciphertext encrypted under $pk_{\mathsf{nd}*}$.

Cryptographic operations by dvc require that dvc use a ticket τ and authorization record $\mathsf{authrec}_{\tau}$ to induce the creation of a logical server at the node nd able to decrypt τ for processing requests bearing the ticket τ; we denote this server by svr_{τ}. (In particular, $\mathsf{svr}* = \mathsf{svr}_{\tau*}$.) nd initializes state for svr_{τ} including a counter $\mathsf{svr}_{\tau}.\mathsf{ctr} \leftarrow 0$ for counting requests bearing τ but reflecting incorrect password guesses.

```
1.  svr_τ.doOperation(req)                    /* can be invoked remotely (by some dvc) */
2.      if (¬fromSameDvc(req, τ))
3.          return ⊥                          /* return if τ and req are not from same device */
4.      if (svr_τ.ctr = q_svr_τ)              /* q_svr_τ is max # of allowed bad password guesses at svr_τ */
5.          return ⊥                          /* return if already at max # of bad password guesses */
6.      if (¬fromSameUser(req, τ))
7.          svr_τ.ctr ← svr_τ.ctr + 1         /* record a bad password guess */
8.          return ⊥                          /* return on bad password guess */
9.      if (req.opType = "sign")
10.         return svr_τ.handleSignReq(req)   /* process sign request and return result */
11.     else
12.         return svr_τ.handleDelReq(req)    /* process delegation request and return result */
```

Figure 1: svr_{τ}.doOperation algorithm adopted from [15]

dvc can then interact with svr_{τ} to either sign messages or delegate. To do so, dvc generates a request req as a function of π and the secret stored in $\mathsf{authrec}_{\tau}$, as well as the request parameters: The message *m* to be signed if a signature operation, or the identity and public key $pk_{\mathsf{nd}'}$ of nd′ if delegating to nd′. dvc then invokes svr_{τ}.doOperation(req), which proceeds as in Figure 1. As shown, svr_{τ} first determines

if: req and τ were created using different dvc secrets (line 2 of Figure 1); the password mistype counter svr_τ.ctr is already at its maximum, q_{svr_τ} (line 4); or req and τ were created using different passwords (line 6). If any of these conditions occur, the request is aborted (lines 3, 5, and 8). Otherwise the request is processed according to the req.opType field (which is a string constant, either **`sign`** or **`delegate`**) and a response is returned (lines 9–12).

If a signature operation, dvc completes the signature for *m* upon receiving a valid response from svr_τ. If a delegation to nd′, the response enables dvc to generate a ticket τ' encrypted under $pk_{\mathsf{nd}'}$. In this case, the inner ticket ζ' is generated by svr_τ and sent to dvc for inclusion in τ'. ζ' is used to convey secret information from svr_τ to the yet-to-be-created $\mathsf{svr}_{\tau'}$.

4.1 Security

We say that svr_τ is *authorized at time t* if either (i) $\tau = \tau^*$ or (ii) at some $t' < t$ and before dvc.comp, dvc performs dvc.delegate(svr′,nd) with a svr′ authorized at time t' to obtain output $\langle \tau, \mathsf{authrec}_\tau \rangle$, and no dvc.revoke($\mathsf{svr}_\tau$) occurs before *t*. In (ii), svr′ is the *consenting server*. In contrast to [15], svr* is always authorized by (i). We motivate this in Section 5.

We divide attackers into four nonoverlapping classes, based on what they compromise and when. We assume an attacker falls into one of these classes non-adaptively, i.e., it does not change its behavior relative to these classes depending on system execution.

A1. An A1 attacker does not compromise dvc.
A2. An A2 attacker compromises dvc, does not compromise π, and compromises no server authorized at the time of dvc.comp.
A3. An A3 attacker compromises dvc, does not compromise π, and compromises some server authorized at the time of dvc.comp.
A4. An A4 attacker compromises both dvc and π, but does not compromise any admissible node.

The security goals achieved in [15] against these attackers are as follows:

G1. An A1 attacker is unable to forge signatures for dvc.
G2. An A2 attacker can forge signatures for dvc with probability at most $q/|\mathsf{D}|$, where *q* is the total number of queries to authorized servers after dvc.comp, and D is the dictionary from which the password is drawn (assumed uniformly at random).
G3. An A3 attacker can forge signatures only if it succeeds in an offline dictionary attack on the password.
G4. An A4 attacker can forge signatures only until all admissible nodes are disabled for dvc.

These properties can be more intuitively stated as follows. If an attacker does not capture dvc (A1), then the attacker gains no ability to forge for the device (G1). On the other extreme, if an attacker captures both dvc and π (A4)—and thus is indistinguishable from the user—but does not compromise any admissible nodes, then it can forge only until all admissible nodes are disabled (G4). The "middle" cases are if the attacker compromises dvc and not π. If it compromises dvc and no then-authorized server is ever compromised (A2), then the attacker can do no better than an online dictionary attack against π (G2). If, on the other hand, when dvc is compromised some authorized server is eventually compromised (A3), then the attacker can do no better than an offline attack against π (G3).

5 Goals

As motivated in Section 1, our high-level goal for coordinating capture protection servers is to improve G2 and G4 from Section 4 (while keeping G1 and G3 unchanged). First we motivate our improvements to G2. This property bounds the probability that an A2 attacker can forge signatures for the device, as a function of the total number q of password queries that the attacker can make to authorized servers after capturing the device. In a straightforward implementation, each server svr would individually limit the number of guesses to some number q_{svr}, and refuse to respond once svr has received q_{svr} queries from dvc with the wrong password. In this case, if A is the set of authorized servers when dvc is captured, then the number of queries that the attacker can make is $q = \Sigma_{\mathsf{svr}\in A}\, q_{\mathsf{svr}}$. Since servers are authorized dynamically, G2 provides little assurance without an additional mechanism to bound q, i.e., while q_{svr} is limited, q may not be. So, one goal is to regain the ability to limit q explicitly:

G2+. An A2 attacker can forge signatures for dvc with probability at most $\hat{q}/|\mathsf{D}|$, where $\hat{q}$ is a prespecified constant and D is the dictionary from which the password is drawn.

Our second goal pertains to G4. As already noted, the number and identity of admissible nodes is not required to be fixed, and it seems most advantageous for it to be specified more fluidly (e.g., "all nodes certified by one of these three certification authorities"). Thus, disabling all admissible nodes, as required in G4, is a challenge. Even if the set of admissible nodes could be determined, disabling each of them may require interacting with potentially hundreds of far-flung nodes all over the world. Therefore, a second goal that we adopt here is to remedy this problem, by making one successful disable operation at nd* imply that dvc is disabled at *all* admissible nodes:

G4+. An A4 attacker can forge signatures only until the time at which nd* is disabled for dvc.

6 Design

Our strategy for achieving G2+ and G4+ is to maintain a shared counter ctr for dvc that records the number of incorrect password guesses made globally against dvc. A server can access this counter using three operations: read, increment, and maximize; see Figure 2. Intuitively, read enables a server to read the current value of ctr, so that the server can refuse to interact with dvc if ctr = $\hat{q}$ thus enforcing G2+. Upon an unsuccessful password guess from dvc, a server will increment ctr. In addition, when nd* is disabled for dvc it invokes maximize to set ctr to $\hat{q}$, and then no server will respond to dvc, enforcing G4+.

```
svr_τ.read()                    svr_τ.increment()                svr_τ.maximize()
    svr_τ.retrieve()                svr_τ.retrieve()                svr_τ.retrieve()
(*) c ← svr_τ.ctr               (*) svr_τ.ctr ← svr_τ.ctr + 1   (*) svr_τ.ctr ← q̂
    V(svr_τ.sem_1)                  V(svr_τ.sem_1)                  V(svr_τ.sem_1)
    return c
```

Figure 2: svr_τ.read, svr_τ.increment and svr_τ.maximize algorithms; can be invoked only locally

6.1 Mutually exclusive access

We strive to support concurrent requests for a counter from multiple servers to allow disabling a compromised device while the attacker is using it, and to permit maximum flexibility in legitimate uses of the device's private key (e.g., device cloning). To ensure the counter's consistency, our implementation enforces mutually exclusive access.

```
1.  svr_τ.initialize()
2.      svr_τ.parent ← τ.getConsentingSvr()     /* extract consenting server from ticket */
3.      svr_τ.children ← ∅
4.      svr_τ.sem_2 ← 1                         /* so that it doesn't block to start with */
5.      if svr_τ.parent = 0                     /* τ = τ* */
6.          svr_τ.arrow ← svr_τ
7.          svr_τ.ctr ← 0                       /* initialize counter */
8.          svr_τ.sem_1 ← 1                     /* allow incoming retrieve requests */
9.      else
10.         svr_τ.arrow ← svr_τ.parent          /* initialize arrow to point to parent */
11.         svr_τ.sem_1 ← 0                     /* don't have the counter; block incoming retrieve requests */
```

Figure 3: svr_τ.initialize algorithm; invoked locally by the node hosting svr_τ

The protocol we propose for ensuring mutually exclusive access consists mainly of the svr_τ.initialize and svr_τ.retrieve functions shown in Figure 3 and Figure 4. svr_τ.initialize is invoked by a node when τ is first submitted to it, and svr_τ.retrieve can be invoked either by svr_τ itself or remotely by another server. In a nutshell, each authorized capture protection server maintains a pointer—here called an *arrow* and denoted svr_τ.arrow—to the server from which it received the last *request* for access to the counter. That is, if

svr_τ receives a request for the counter, then svr_τ requests it from $svr' \leftarrow svr_\tau.arrow$ (line 4 in Figure 4) by invoking $svr'.retrieve()$ (line 12 or 15). It also sets $svr_\tau.arrow$ to be the identity of the requester, denoted caller in Figure 4 (line 5). caller is authenticated by means that will be discussed in Section 6.2, so that if caller = svr'' and svr'' is not compromised, then svr'' performed this method invocation. Upon receiving the counter in response to the $svr'.retrieve()$ request, svr returns the counter to caller (line 17). Figure 5 shows the effects of a retrieve request initiated by a server svr.

```
1.  svr_τ.retrieve()                                  /* caller is id of invoking server */
2.      if caller ∉ svr_τ.children ∪ {svr_τ.parent, svr_τ}   /* svr_τ.children, svr_τ.parent described in Sec. 5.2 */
3.          return ⊥                                  /* ignore requests from unknowns */
4.      svr' ← svr_τ.arrow                            /* see who most recently requested the counter */
5.      svr_τ.arrow ← caller                          /* record our caller as last requesting the counter */
6.      if (svr' = svr_τ)                             /* if I most recently requested the counter, then ... */
7.          P(svr_τ.sem_2)                            /* ... block request until I complete previous ones, */
8.          P(svr_τ.sem_1)                            /* ... block caller until I'm done with the counter, */
9.          V(svr_τ.sem_2)                            /* ... and permit requests to make progress again */
10.     else if (caller = svr_τ)                      /* if I am the one requesting, then ... */
11.         P(svr_τ.sem_2)                            /* ... block request until I complete previous ones, */
12.         svr_τ.ctr ← svr'.retrieve()               /* ... remote call to retrieve counter (blocks thread), */
13.         V(svr_τ.sem_2)                            /* ... and permit requests to make progress again */
14.     else                                          /* I am just a "transit server" for this request */
15.         svr_τ.ctr ← svr'.retrieve()
16.     if (caller ≠ svr_τ)
17.         return svr_τ.ctr                          /* return counter to the remote caller */
```

Figure 4: svr_τ.retrieve algorithm; can be invoked locally or remotely (by another server)

We emphasize that $svr_\tau.retrieve()$ may be invoked concurrently, e.g., by multiple remote servers. For simplicity, the psuedocode of Figure 4 and subsequent figures assumes that a thread of execution runs atomically (i.e., non-preemptively, without interference from other threads in svr_τ) until completion or until it blocks either on a semaphore[2] (line 7, 8 or 11) or due to invoking retrieve on another server (line 12 or 15). Once a running thread blocks, another can enter a retrieve operation. We denote global variables accessible to all threads using the "svr_τ." prefix, e.g., $svr_\tau.arrow$. ("svr_τ", i.e., the identity of this server, is also global.) Variables without this prefix, specifically svr' and caller, are local to this thread.

Use of two different semaphores requires some explanation. $svr_\tau.sem_1$ is used to ensure that once the counter is retrieved by svr_τ, requests to pull the counter away are blocked until svr_τ has executed its critical section (the lines marked "(*)" in Figure 2). $svr_\tau.sem_2$ is used to block any retrieve requests made by svr_τ until svr_τ services the retrieve requests it received previously from others. Starvation is avoided if the retrieve requests blocked on each semaphore are serviced in a first-in-first out order per V($svr_\tau.sem_i$) invocation.

[2] To remind the reader, a semaphore *s* is a concurrency control primitive that represents a non-negative integer counter with two atomic operations: V(*s*) increments s by one; P(*s*) blocks the calling thread while *s*=0 and then decrements *s* by one [9].

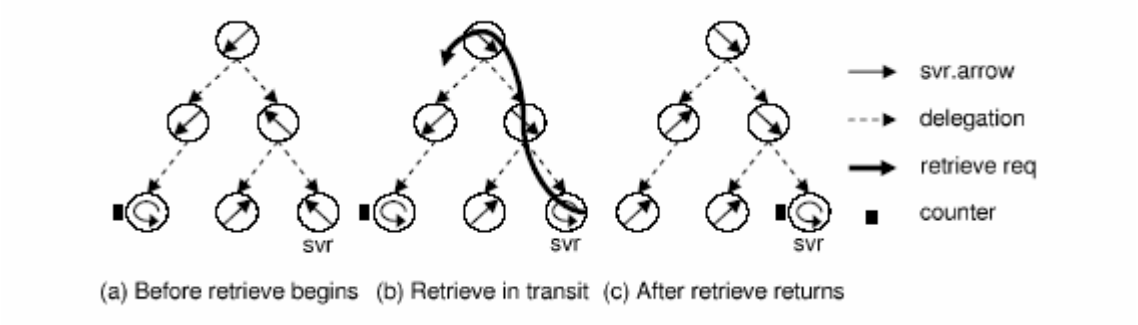

Figure 5: retrieve initiated by svr

6.2 Limiting counter access

To achieve G2+ and G4+, it is necessary that only uncompromised servers can pass the counter once dvc is captured; otherwise a compromised server could manipulate the counter. For an A4 attacker, it would suffice to permit only admissible nodes to pass the counter, since admissible nodes are uncompromised by assumption. However, for an A2 attacker, where admissible nodes may be compromised, this simple rule does not suffice. Fortunately, since the servers authorized when an A2 attacker captures dvc are never compromised (by the definition of A2), it suffices to permit only authorized servers to pass or hold the counter. Because authorized servers are hosted only on admissible nodes, this is consistent with the A4 case.

Our protocol thus restricts counter passing to occur only between authorized servers in the A2 case. This is complicated by the fact that the set of authorized servers is dynamic, and there is no trustworthy record of this set. This problem can be partially alleviated by having a consenting server svr_τ record all the servers it has consented to authorize in a local set svr_τ.children (see line 2 of Figure 4). For simplicity, we portray svr_τ.children as a set of server names in our figures, though in reality a different representation is required. Specifically, because the ticket τ' resulting from a delegation to which svr_τ consented is not known to svr_τ, svr_τ cannot explicitly include τ' in svr_τ.children. However, if svr_τ includes a new cryptographic key k within both svr_τ.children and the inner ticket ζ' that it contributes as an input to the creation of τ', then svr_τ can use k to authenticate requests from $\mathsf{svr}_{\tau'}$. For reasons described in Section 6.3, svr_τ also must send a preimage resistant and collision resistant hash of k, h_k, to dvc for storage in $\mathsf{authrec}_{\tau'}$.

To ensure that the counter is passed only between authorized servers, it is also necessary for $\mathsf{svr}_{\tau'} \in \mathsf{svr}_\tau$.children to authenticate retrieve requests from svr_τ. Fortunately, $\mathsf{svr}_{\tau'}$ can use the key k inserted into ζ' above (or another) to authenticate communication from svr_τ. To facilitate $\mathsf{svr}_{\tau'}$ contacting svr_τ the first time, svr_τ's address is included within ζ' and τ'; $\mathsf{svr}_{\tau'}$ assigns this address to $\mathsf{svr}_{\tau'}$.parent (line 2 of Figure 3). (For $\tau_{\mathsf{svr}*}$, a predetermined constant 0 is inserted in place of the consenting server address.)

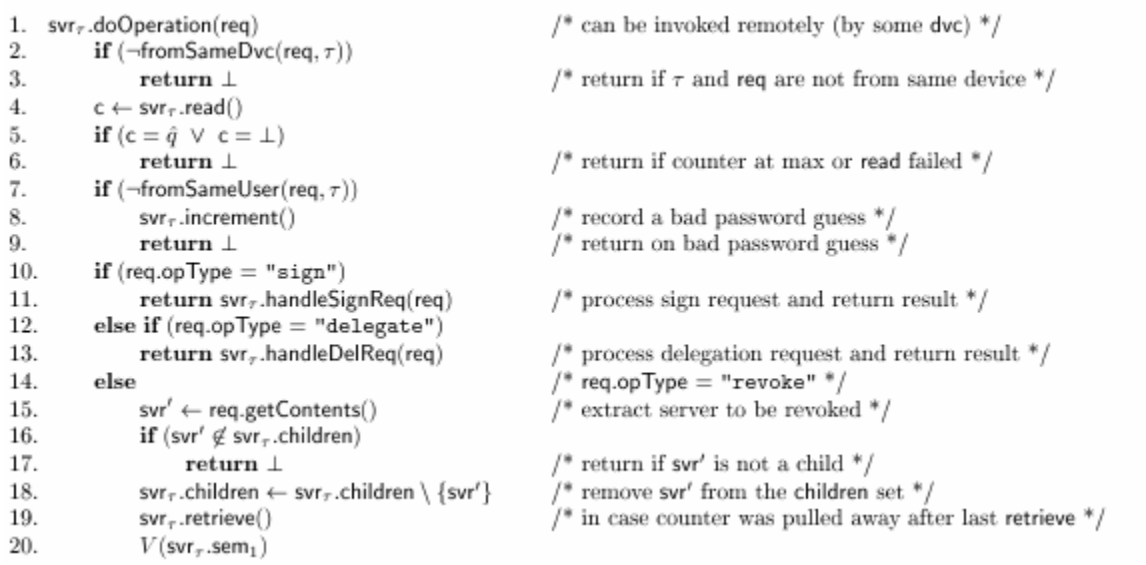

Figure 6: New svr$_\tau$.doOperation algorithm

6.3 Revocation

In Section 6.2, we described mechanisms by which servers keep track, in their children and parent variables, of other authorized servers (and how to authenticate them). Because of this, we must extend revocation to update these variables, so that servers do not work with outdated information about which servers are authorized. Whereas initially revocation was an operation local to dvc [15], here we extend it to include interaction with a server to update its children variable.

Specifically, before revoking a server svr$_{\tau'}$, dvc informs svr$_{\tau'}$.parent of this revocation. This notification indicates that dvc plans to revoke not just svr$_{\tau'}$ but also the servers in svr$_{\tau'}$.children, their children, and so forth. The purpose of dvc revoking the entire set of delegations derived from svr$_{\tau'}$ is to ensure that all still-authorized servers can continue to access the counter for dvc. Doing otherwise could partition the tree of delegations, and the counter may become inaccessible for some authorized servers. Note that svr* is never part of this revoked component. This is required since to achieve G4+, svr* must be able to retrieve the counter.

During revocation of svr$_{\tau'}$, dvc informs svr$_\tau$ = svr$_{\tau'}$.parent by issuing a request req (with req.opType = **revoke**) to svr$_\tau$. The revoked server svr$_{\tau'}$ is identified in req by h_k (hash of the key k) that svr$_\tau$ sent to dvc during the delegation protocol; see Section 6.2. This identifier for svr$_{\tau'}$ is extracted via req.getContents() (line 15 of Figure 6). This request induces a removal of svr$_{\tau'}$ (or rather k) from svr$_\tau$.children. Also note that svr$_\tau$ retrieves the counter (see line 19 of Figure 6) thereby ensuring that the counter is not lost when svr$_{\tau'}$ is revoked. svr$_\tau$ retrieves the counter after removing svr$_{\tau'}$ from svr$_\tau$.children so that

any subsequent requests from $\mathsf{svr}_{\tau'}$ to retrieve the counter are rejected. After this svr_{τ}.doOperation call, dvc deletes $\mathsf{authrec}_{\tau'}$, i.e., performs dvc.revoke($\mathsf{svr}_{\tau'}$). Moreover, it must invoke dvc.revoke($\mathsf{svr}_{\tau''}$) for each $\mathsf{svr}_{\tau''} \in \mathsf{svr}_{\tau'}$.children, their children, and so on.

6.4 Disabling

To achieve G4+ we require that nd* can set the counter value to its maximum value $\hat{q}$ so as to disable dvc at all admissible nodes. The revocation mechanism presented in Section 6.3 ensures that svr* can always request the counter. Hence, upon receiving a disable request, nd* performs a svr*.maximize operation that causes servers to stop responding to dvc. The disable algorithm also uses a capability to authenticate the disable request; see [15].

Note that the nd.disable request can also be sent to another node nd hosting an authorized server for dvc. However, the ability of an A4 attacker to revoke and delegate makes it impractical to locate nodes besides nd* to disable after dvc has been compromised. Though the attacker can perform a dvc.revoke(svr*) operation, this will not restrict svr*'s access to the counter due to the measures described in Section 6.3. Hence, nd* is able to complete a disable request.

7 Implementation in Fleet

In Sections 3–6, we presented our algorithms for coordinating capture protection servers, treating each capture protection server as a non-replicated object. However, the coordination protocols we presented, while improving some types of protection (see Section 5), do exacerbate the effects of a benign or malicious capture-protection server failure, in that such a failure could prevent any server from being able to retrieve the counter or, therefore, from assisting the device in any cryptographic operations. Consider nd*, for example: In order for property G4+, to be useful, it is necessary that a client can disable dvc at nd*, which requires nd* to be available. More generally, if svr is down and another server invokes svr.retrieve(), then it is possible that all subsequent cryptographic operations by dvc, performed using any capture protection server, will block at least until svr recovers. It is thus necessary in practice that such a protocol be built in a way that ensures the survivability of each capture protection server.

Fortunately, the Fleet system, while being the primary beneficiary of capture protection in this effort, also provides a utility for building such survivable services [16]. Moreover, integration of this capture-protection infrastructure with the Fleet system is fairly straightforward; in principle, Fleet enables survivable implementations of arbitrary objects, of which a capture-protection server is one example; though it did require adaptations to both the capture-protection infrastructure and Fleet, which we describe below.

First, in our implementation, each capture protection server svr is implemented as a Fleet object replicated across several nodes (each running the Fleet server-side software). As a result, it is no longer possible to simply designate pk_{svr} to be pk_{nd} for the single nd hosting svr (see Section 3); there are now several such nodes. Adapting the protocols to accommodate this, however, is a simple matter, by creating a new public key/secret key pair (pk_{svr}, sk_{svr}) for svr and delivering sk_{svr} to each node nd participating in the implementation of svr, encrypted under pk_{nd}, upon creation of svr.

A second consequence of replicating svr is that svr must be implemented deterministically, since the form of replication supported in Fleet that is appropriate for this application is one in which all replicas are deterministic (see [16]). In order to support this, we modified the capture protection server implementation to replace random choices in each method invocation using a pseudorandom function keyed with (a cryptographic hash of) sk_{svr} and applied to method arguments. Provided that the pseudorandom function is secure (indistinguishable from a random function to those not having the key), then this results in no significant loss to security.

Finally, we were required to modify Fleet itself to support this application, since when one (replicated) capture protection server calls another (e.g., to retrieve the counter), this involves a replicated object invoking a method on another replicated object, which was not transparently supported in previous versions of Fleet [16]. To accomplish this, we adapted the method invocation protocol from [7] to better support replicated clients.

Aside from these adaptations, the implementation of capture protection in Fleet closely reflects the description in Section 6.

8 Summary

In this report we detailed a client capture protection infrastructure for use within the context of Fleet, a survivable object store. The innovation in this work is a simple data-sharing protocol, for capture-protection servers, that strictly limits online dictionary attacks on a client device that is captured, and that achieves immediate disabling of the client device even with dynamically changing server populations.

The capture-protection infrastructure described in this report forms a symbiotic relationship with Fleet. On one hand, the capture-protection infrastructure substantially hardens Fleet against an important class of attack, in which a user-driven client device with authority to invoke method invocations on Fleet objects is captured. This infrastructure prevents or substantially limits the damage that such an attacker can inflict. On the other hand, the availability of the algorithms in this infrastructure—and thus the availability of the client device's private key operations—is particularly vulnerable to even the benign failure of a capture-protection server. So, implementing each capture-protection server as a survivable Fleet object is central to the client's availability.

9 References

[1] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems* 9(1):1-20, Feb. 1991.

[2] D. Boneh, X. Ding, G. Tsudik, and M. Wong. A method for fast revocation of public key certificates and security capabilities. In *Proc. 10th USENIX Security Symposium*, pages 297-308, August 2001.

[3] R. Baldoni, A. Virgillito, R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. In *Proceedings of the 7th IEEE Symposium on Computers and Communications*, July 2002.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[5] Y. Chang, M. Singhal and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proc. 9th IEEE Symposium on Reliable Distributed Systems*, pages 146-154, 1990.

[6] Y. Chen and J. L. Welch. Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks. In *Proceedings of the 6th International Workshop on Discrete Algorithms & Methods for Mobile Computing & Communications*, September 2002.

[7] G. Chockler, D. Malkhi and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 11–20, April 2001.

[8] M. J. Demmer and M. P. Herlihy. The Arrow distributed directory protocol. In *Proc. 12th International Symposium on Distributed Computing*, pp. 119-133, 1998.

[9] E. W. Dijkstra. Cooperating sequential processes. Mathematics Department, Technological University, Eindhoven, The Netherlands, 1965.

[10] X. Ding, D. Mazzocchi, and G. Tsudik. Experimenting with server-aided signatures. In *Proceedings of the 2002 ISOC Network & Distributed System Security Symposium*, February 2002.

[11] R. Ganesan. Yaksha: Augmenting Kerberos with public key cryptography. In *Proceedings of the 1995 ISOC Network & Distributed System Security Symposium*, pages 132–143, February 1995.

[12] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security* 4(4), November 2001.

[13] L. Lamport, R. Shostak, M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382–401, July 1982.

[14] P. MacKenzie and M. K. Reiter. Networked cryptographic devices resilient to capture. *International Journal of Information Security* 2(1):1–20, November 2003.

[15] P. MacKenzie and M. K. Reiter. Delegation of cryptographic servers for capture-resilient devices. *Distributed Computing* 16(4):307–327, December 2003.

[16] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the Fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition* (DISCEX II), Vol. II, pages 126–136, June 2001.

[17] J. Menezes, P. C. van Oorschot and S. A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.

[18] M. L. Neilsen and M. Mizuno. A dag-based algorithm for distributed mutual exclusion. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 354–360, 1991.

[19] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computing Syst*ems 7(1):61–77, February 1989.

[20] R. L. N. Reddy, B. Gupta and P. K. Srimani. A new fault tolerant distributed mutual exclusion algorithm. In *Proceedings of the ACM Symposium on Applied Computing*, pages 831–839, 1992.

[21] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

[22] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pages 99–110, Springer-Verlag, 1995.

[23] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21(2):120–126, February 1978.

[24] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys* 22(4):299–319, December 1990.

[25] J. E. Walter, J. L. Welch and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. In *Proceedings of the 2nd International Workshop on Discrete Algorithms & Methods for Mobile Computing & Communications*, October 1998.